Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н. Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К КУРСОВОЙ РАБОТЕ
НА ТЕМУ:
Аутентификация операционной системы с помощью USB-ключа

Студент ИУ7-74Б _____ Керимов А. Ш.
(Группа) (Подпись, дата) (Фамилия И. О.)

Руководитель курсовой работы _____ Тассов К. Л.
(Подпись, дата) (Фамилия И. О.)

2020 г.

Оглавление

Введение

В современном мире остро стоит вопрос обеспечения эффективной защиты информации. В основном, у людей множество важной информации содержится в персональном компьютере, доступ к которому осуществляется, чаще всего, с помощью проверки правильности пароля.

Такой способ защиты информации не лишён недостатков: пароль к компьютеру можно узнать различными методами. Более безопасным вариантом контроля доступа к компьютеру является многофакторная аутентификация, в частности, с дополнительной проверкой подключённого USB-ключа.

Курсового работа посвящена разработке программного обеспечения, которое добавляет аутентификацию на основе проверки серийного номера подключённого USB-устройства в процесс загрузки ОС Linux.

1 Аналитический раздел

1.1 Постановка задачи

В рамках выполнения курсовой работы необходимо решить следующие **задачи**:

- формализовать задание в виде определения необходимого функционала;
- провести анализ методов реализации;
- спроектировать программное обеспечение;
- реализовать спроектированное программное обеспечение.

1.2 Формализация задания

В соответствии с техническим заданием на курсовую работу необходимо разработать программное обеспечение, которое добавляет проверку наличия подключённого через интерфейс USB flash-накопителя с заданным серийным номером в процесс загрузки операционной системы Linux.

Ограничение: использование ПК подразумевается одним пользователем с единственным usb-ключём.

1.3 Анализ методов реализации

- PAM
- LUKS
- Патч ядра
- Загружаемый модуль ядра

1.3.1 PAM

PAM (**P**luggable **A**uthentication **M**odules, *рус.* **П**одключаемые **М**одули **А**утентификации, ПМА) — это набор разделяемых библиотек, позволяющих разрабатывать программы, не зависящие от схемы аутентификации.

На заре UNIX аутентификация пользователя выполнялась следующим образом: программа `login` требовала ввести имя и пароль, затем этот пароль шифровался известным алгоритмом и сравнивался с соответствующей зашифрованной записью в файле `/etc/passwd`.

Впоследствии программа `login` была переписана, т. к. зашифрованный пароль перестал открыто храниться в `/etc/passwd`, и был перемещён в отдельный файл скрытых паролей `/etc/shadow`, доступный для чтения только суперпользователю.

Появлялись новые схемы аутентификации, и проблема заключалась в том, что каждый раз приходилось переписывать все необходимые программы (`login`, `dtlogin`, `rlogin`, `ftp`, `rsh`, `telnet`, ...).

Для отделения программ от механизма аутентификации в октябре 1995 года инженерами американской компании Sun Microsystems был предложен PAM [1]. Впервые же PAM появился в августе 1996 года в Red Hat Linux 3.0.4 (Rembrand II) [2].

Конфигурационные файлы PAM для различных приложений располагаются в директории `/etc/pam.d/`. Имя каждого конфигурационного файла в этой директории совпадает с именем программы, для которого он предназначен.

Конфигурационные файлы описывают стек модулей. Дескриптор модуля имеет формат:

```
<Тип-Модуля> <Флаг-Управления> <Путь-К-Модулю> [Параметры-Модуля]
```

`Тип-Модуля` может быть одним из четырёх, перечисленных ниже:

- `auth` — модули аутентификации. Предназначены для аутентификации и идентификации пользователей, а также создания, обновления и уничтожения учётных данных.

- `account` — модули учётных записей. Предназначены для проверки срока действия паролей, учётных записей и ограничения времени

доступа. После того, как пользователь идентифицирован с помощью модулей аутентификации, модули учетных записей определяют, следует ли предоставить пользователю доступ.

- `session` — модули сеансов. Предназначены для управления выполнением задач во время начала и окончания сессии аутентификации.
- `password` — модули паролей. Предназначены для изменения паролей.

`Флаг-Управления` определяет дальнейший порядок исполнения стека при удачном или неудачном прохождении соответствующего модуля. Наиболее распространённые флаги управления:

- `required` — модуль должен завершиться успешно. В случае неудачи статус ошибки будет возвращён после исполнения оставшихся модулей.
- `requisite` — модуль должен завершиться успешно. В случае неудачи статус ошибки будет возвращён сразу.
- `sufficient` — успешное завершение модуля возвращает статус `PAM_SUCCESS` сразу. Неудача не препятствует успешному исполнению стека.
- `optional` — не оказывает влияния на статус исполнения стека.

Проверку серийного номера подключённого USB-устройства необходимо реализовать в виде отдельного PAM модуля, дескриптор которого с флагом управления `requisite` поместить в конфигурационный файл `/etc/pam.d/login` перед непосредственной проверкой пароля.

1.3.2 LUKS

LUKS (**L**inux **U**nified **K**ey **S**etup) — это спецификация форматов шифрования дисков, предназначенная для Linux, созданная Клеменсом Фрувиртом в 2005 году [3].

В то время как большая часть программного обеспечения для шифрования дисков реализует различные, несовместимые и недокументированные

форматы, LUKS реализует платформенно-независимый стандартный дисковый формат для использования в различных инструментах. Это не только облегчает совместимость и взаимодействие между различными программами, но также гарантирует, что все они реализуют управление паролями безопасным и документированным способом.

Для реализации двухфакторной аутентификации с использованием LUKS можно зашифровать раздел диска с помощью файла-ключа, помещённого на USB-накопитель и содержащего либо сам серийный номер, либо хеш от него.

1.3.3 Патч ядра

Один из способов реализации поставленной задачи заключается в изменении кода ядра непосредственно.

Для того, чтобы добавить патч, необходимо найти правильное место в ядре для вставки кода.

При подключении устройства через интерфейс USB, в системном журнале появляются сообщения вида:

```
New USB device found, idVendor=%04x, idProduct=%04x, bcdDevice=%2x.%02x
New USB device strings: Mfr=%d, Product=%d, SerialNumber=%d
```

Путём поиска в исходном коде ядра, находим функцию, которая производит записи такого формата в системный журнал:

```
/* inside drivers/usb/core/hub.c */
static void announce_device(struct usb_device *udev);
```

Так как функция `announce_device`[1] является лишь обёрткой для записи в системный журнал, необходимо проанализировать функции, которые её вызывают.

Вызов функции `announce_device` происходит в единственном месте — в функции `usb_new_device`[2]:

[1] https://elixir.bootlin.com/linux/v5.9.14/source/drivers/usb/core/hub.c#L2271

[2] https://elixir.bootlin.com/linux/v5.9.14/source/drivers/usb/core/hub.c#L2500

```
/* inside drivers/usb/core/hub.c */
int usb_new_device(struct usb_device *udev);
```

Описание функции ядра `usb_new_device` свидетельствует о том, что именно она вызывается при подключении USB-устройства, следовательно, проверку серийного номера необходимо добавить в эту функцию.

1.3.4 Загружаемый модуль ядра

Загружаемый модуль ядра (*англ.* **L**oadable **K**ernel **M**odule, LKM) — это объектный файл, содержащий код, расширяющий возможности ядра операционной системы.

Загружаемые модули позволяют вносить изменения в код ядра без его перекомпиляции. При этом в распоряжении разработчика оказываются все имеющиеся ядрёные библиотеки.

Для реализации поставленной задачи необходимо зарегистрировать собственный USB-драйвер, а именно:

- заполнить дескриптора драйвера — структуру `usb_driver`;
- непосредственно зарегистрировать структуру в системе с помощью функции `usb_register` [4].

Рассмотрим наиболее важные поля структуры `usb_driver` (листинг 1.1).

Листинг 1.1: `struct usb_driver`

```
1  #include <linux/usb.h>
2
3  struct usb_driver {
4          const char *name;
5
6          int (*probe) (struct usb_interface *intf,
7                        const struct usb_device_id *id);
8
9          void (*disconnect) (struct usb_interface *intf);
10
11         const struct usb_device_id *id_table;
```

```
12
13          // ...
14 };
```

- `name` — имя драйвера. Должно быть уникальным среди драйверов USB. Обычно совпадает с именем модуля.
- `id_table` — это массив структур `usb_device_id`. Определяет список устройств, при подключении которых вызывается код драйвера. Массив заканчивается терминирующей пустой структурой. В случае отсутствия в массиве структур, система будет пытаться подключить каждое устройство к драйверу.
- `probe` и `disconnect` — это callback-функции, которые вызываются системой при подключении и отключении USB-устройства.

Таким образом, проверку серийного номера подключённого USB-устройства необходимо произвести в функции `probe` разрабатываемого модуля ядра.

1.3.5 Выбор метода реализации

Наиболее правильным было бы использовать PAM, т. к. именно для решения задач, связанных с аутентификацией, они и создавались.

Однако использование готовых инструментов, таких как PAM или LUKS, в меньшей степени способствует приобретению практических навыков реализации задач системного программирования.

В связи с этим, в качестве метода из двух оставшихся выбрана разработка загружаемого модуля, поскольку патч ядра уместен лишь в случае невозможности решения задач средствами lkm.

Вывод

В результате проведённого анализа методов реализации аутентификации ОС Linux с помощью USB-ключа была выбрана разработка загружае-

мого модуля ядра.

2 Конструкторский раздел

2.1 Проектирование загружаемого модуля ядра

При загрузке модуль ядра не получает никаких параметров: для обеспечения большей безопасности серийный номер USB-ключа зашит в коде.

Очевидно, модуль ядра будет загружаться при запуске операционной системы. При этом проверку серийного номера USB-устройства необходимо произвести до аутентификации с помощью пароля.

В Linux программа, которая управляет командной строкой и соответственно подключённым терминалом, называется `agetty`. Её цель — защитить систему от несанкционированного доступа. Большинство времени процесс `agetty` находится в состоянии прерываемого сна (`TASK_INTERRUPTIBLE`) в ожидании ввода. Одним из способов приостановления возможности аутентификации пользователя с помощью пароля является переключение процесса `agetty` в состояние непрерываемого сна (`TASK_UNINTERRUPITBLE`).

При загрузке модуля происходит регистрация USB-драйвера и запуск потока блокировки `agetty`. В обработчике подключения USB-устройств драйвера выполняется проверка серийного номера и, в случае совпадения, остановка потока блокировки `agetty`. По истечении 30 секунд при отсутствии подключённого USB-ключа процесс блокировки `agetty` завершает работу операционной системы.

2.1.1 Алгоритм установки состояния процесса

Оформим отдельной подпрограммой алгоритм установки состояния процесса (рис. 2.1). Функция возвращает `true`, если процесс с заданным именем был найден и успешно переведён в требуемое состояние. Иначе — `false`.

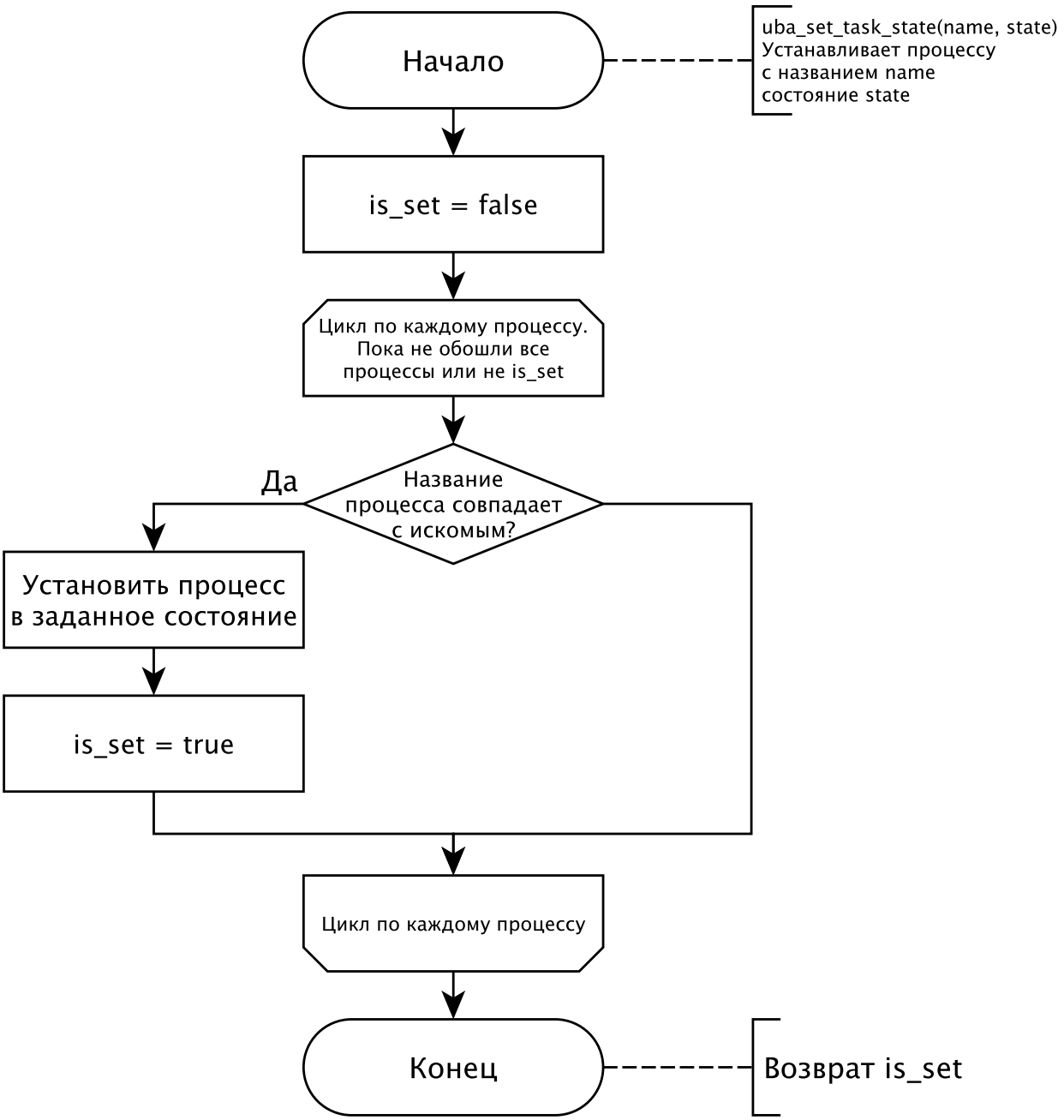

Рис. 2.1: Схема алгоритма установки состояния процесса

2.1.2 Алгоритм функции потока блокировки

На рисунке 2.2 представлена схема алгоритма функции потока блокировки процесса `agetty`.

Сначала процесс `agetty` переключается в состояние непрерываемого сна, затем производится обратный отсчёт до завершения работы системы, остановить который может лишь сигнал об остановке потока.

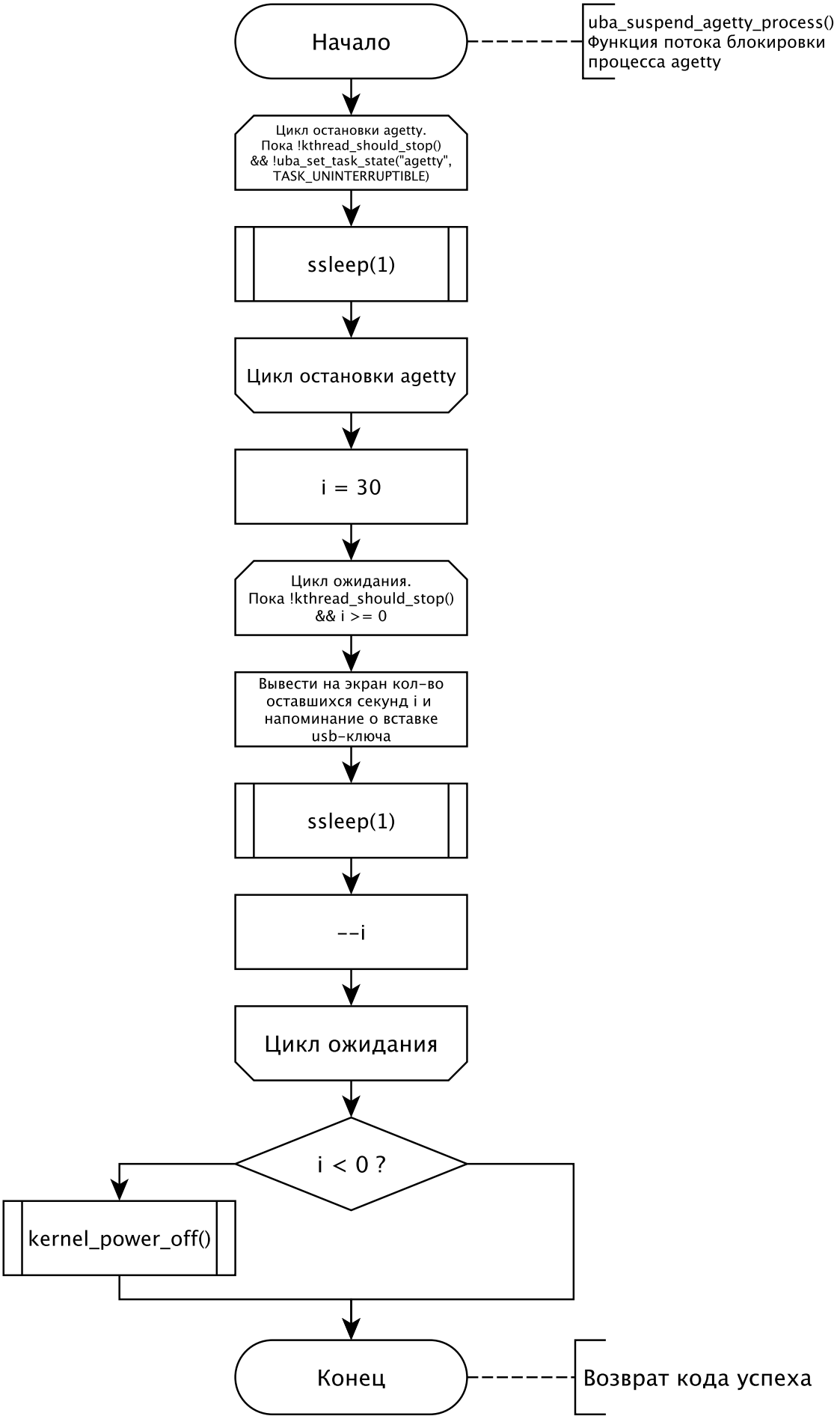

Рис. 2.2: Схема алгоритма функции потока блокировки процесса `agetty`

2.1.3 Алгоритм проверки подлинности ключа

На рисунке 2.3 представлена схема алгоритма проверки подлинности USB-ключа.

После успешной аутентификации дальнейшие проверки подлинности не производятся, процесс `agetty` переводится обратно в состояние прерываемого сна.

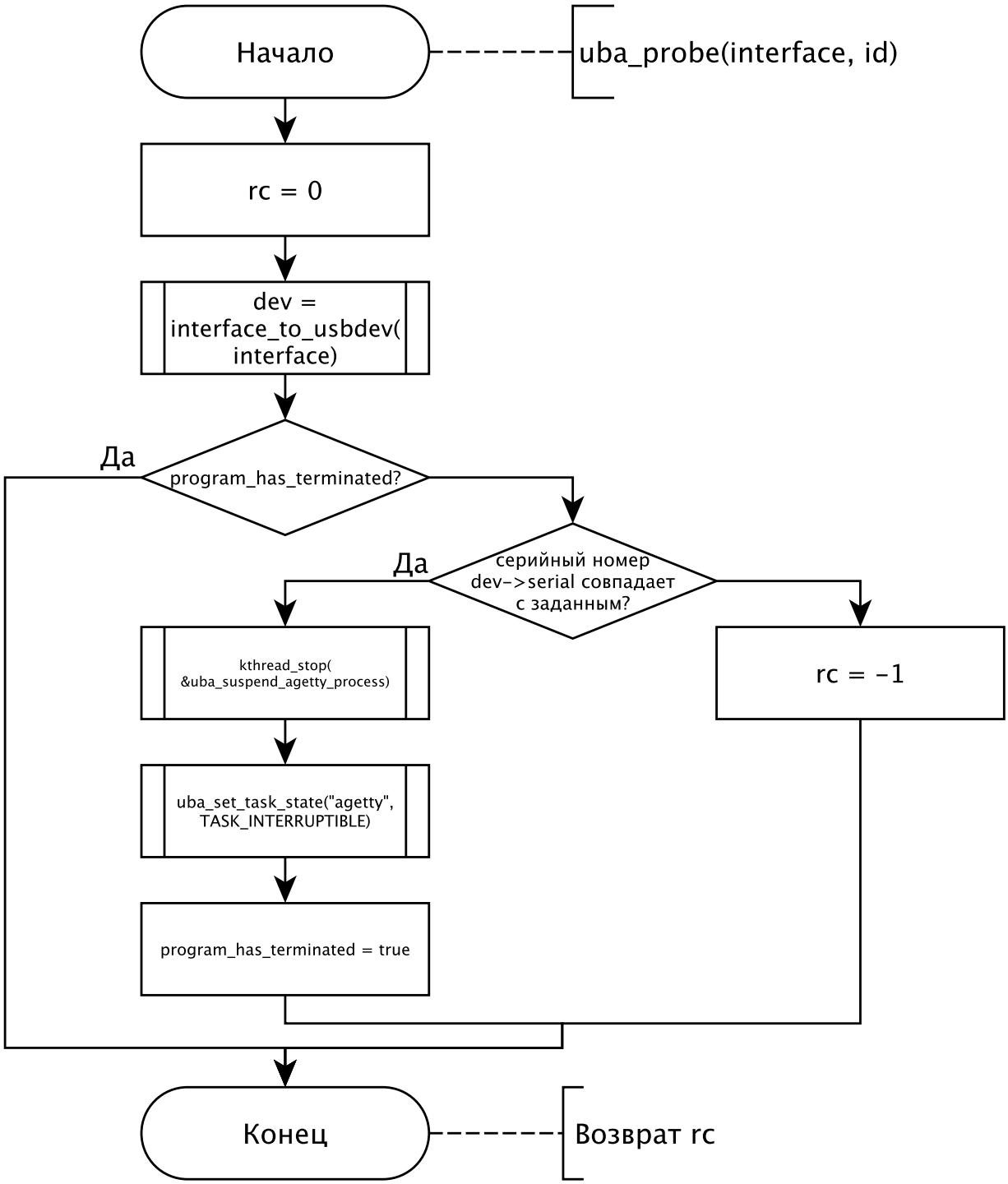

Рис. 2.3: Схема алгоритма проверки подлинности USB-ключа

Вывод

В результате проектирования разработаны алгоритмы проверки подлинности и функции потока блокировки, что позволяет перейти к реализации загружаемого модуля в программном коде.

3 Технологический раздел

3.1 Выбор языка программирования

Перечислим основные особенности, которыми должен обладать язык программирования для разработки загружаемого модуля ядра операционной системы Linux:

- язык высокого уровня;
- широко используется для разработки загружаемых модулей ядра Linux;
- компилируемый;
- модульный;
- быстрый;
- компактный.

Всеми перечисленными особенностями обладает только язык C, поэтому именно он и выбран в качестве языка программирования для разработки загружаемого модуля ядра Linux. Такой выбор не удивителен — язык C первоначально создавался как язык системного программирования. В исходном коде ядра Linux доля исполняемых файлов на языке C составляет приблизительно 97 % [5].

3.2 Выбор среды программирования

Теперь перечислим основные особенности, которыми должна обладать среда программирования для разработки загружаемого модуля ядра ОС Linux.

- текстовый интерфейс;
- компактность;

- быстродействие;
- удобство использования
- широкое распространение среди разработчиков ядра Linux;

Интегрированные среды разработки с графическим интерфейсом не обладают всеми перечисленными особенностями. Поэтому среда программирования выбрана покомпонентно:

- текстовый редактор — vim,
- компилятор — gcc,
- средство автоматизации сборки — make.

3.3 Реализация загружаемого модуля ядра

В листинге 3.3 (приложение А, стр. 22) представлен исходный код загружаемого модуля ядра.

Предварительно необходимо задать собственный серийный номер USB-ключа `UBA_SERIAL`, узнать который можно, например, просмотром системного журнала командой `$ dmesg` после подключения.

3.4 Действия по установке ПО

В листинге 3.4 (приложение В, стр. 25) представлен Makefile загружаемого модуля ядра.

Действия по установке ПО показаны в листинге 3.1.

Листинг 3.1: Установка

```
1 $ make
2 # make enable_at_boot
3 # make boot_in_console_mode
4 # make enable_printing_kernel_journal_on_tty
```

Рассмотрим детальнее каждый шаг.

- Цель по умолчанию (`make` без параметров) производит сборку загружаемого модуля ядра.
- Цель `enable_at_boot` помещает название разработанного модуля в `/etc/modules`, тем самым обеспечивается автоматическая загрузка модуля ядра при запуске операционной системы.
- Цель `boot_in_console_mode` изменяет конфигурационные файлы загрузчика `grub2` и подсистемы инициализации и управления службами `systemd` для запуска операционной системы в консольном режиме.
- Цель `enable_printing_kernel_journal_on_tty` создаёт `rsyslog`-конфиг, позволяющий увидеть сообщения разработанного модуля ядра в терминале.

Выполнение последней цели является необязательным для функционирования программы, однако пропуск этого шага уместен лишь в том случае, если есть необходимость в сокрытии от пользователя информации, объясняющей порядка аутентификации.

К каждой перечисленной цели в Makefile имеется противоположная, действия по удалению ПО показаны в листинге 3.2.

Листинг 3.2: Удаление

```
# make disable_printing_kernel_journal_on_tty
# make boot_in_graphical_mode
# make disable_at_boot
$ make clean
```

При необходимости использования графического интерфейса пользователя после успешного прохождения двухфакторной аутентификации следует ввести

```
$ systemctl isolate graphical.target
```

Вывод

В результате разработки загружаемого модуля ядра получено программное обеспечение, обеспечивающее аутентификацию с помощью USB-ключа.

Заключение

В результате выполнения курсовой работы был разработано программное обеспечение, добавляющее проверку наличия подключённого через интерфейс USB flash-накопителя с заданным серийным номером в процесс загрузки операционной системы Linux.

В процессе выполнения курсовой работы были выполнены все поставленные задачи в полном объеме, а именно:

- формализовано задание, определён необходимый функционал;
- проведён анализ методов реализации;
- спроектировано программное обеспечение;
- реализовано спроектированное программное обеспечение.

Литература

[1] RFC 86.0 — Unified login with Pluggable Authentication Modules (PAM). URL: http://www.opengroup.org/rfc/rfc86.0.html (дата обращения 04.12.2020).

[2] Red Hat Linux — 3.0.4 (Rembrandt II) BETA announcement. URL: https://www.linux.co.cr/distributions/review/1996/0323.html (дата обращения 04.12.2020).

[3] Fruhwirth, C. New Methods in Hard Disk Encryption. URL: https://clemens.endorphin.org/nmihde/nmihde-A4-ds.pdf (дата обращения: 04.12.2020).

[4] Linux Kernel Documentation: Writing USB Device Drivers — URL: https://kernel.readthedocs.io/en/sphinx-samples/writing_usb_driver.html (дата обращения 04.12.2020).

[5] Linux kernel source tree — URL: https://github.com/torvalds/linux (дата обращения: 05.12.2020).

Приложение А. Исходный код модуля

Листинг 3.3: usb_boot_authenticationc.c

```c
#include <linux/init.h>
#include <linux/kthread.h>
#include <linux/module.h>
#include <linux/reboot.h>
#include <linux/sched/signal.h>
#include <linux/usb.h>

#define UBA_MODULE_NAME        "usb_boot_authentication"
#define UBA_MODULE_DESCRIPTION "Hardware authentication for Linux using ordinary USB Flash
    Drives"
#define UBA_MODULE_AUTHOR      "Ahmed Kerimov <kerimov.dev@yandex.ru>"
#define UBA_MODULE_LICENSE     "GPL"

MODULE_DESCRIPTION(UBA_MODULE_DESCRIPTION);
MODULE_AUTHOR(UBA_MODULE_AUTHOR);
MODULE_LICENSE(UBA_MODULE_LICENSE);

static bool program_has_terminated;
static struct task_struct *suspend_agetty_process;

static bool uba_set_task_state(const char *comm, long state)
{
        struct task_struct *task;

        for_each_process(task) {
                if (!strcmp(task->comm, comm)) {
                        task->state = state;
                        return true;
                }
        }

        return false;
}

static void uba_stop_thread(struct task_struct **thread)
{
        kthread_stop(*thread);
        *thread = NULL;
}

static int uba_suspend_agetty_process(void *data)
{
        int i;

        while (!uba_set_task_state("agetty", TASK_UNINTERRUPTIBLE) && !kthread_should_stop()) {
                ssleep(1);
        }

        for (i = 30; i >= 0 && !kthread_should_stop(); --i) {
                printk(KERN_NOTICE "%s: %02d secs to shutdown\n", UBA_MODULE_NAME, i);
                printk(KERN_NOTICE "%s: plug in the USB key to start\n", UBA_MODULE_NAME);
                ssleep(1);
        }
        if (i < 0) {
                kernel_power_off();
```

```
        }

        printk(KERN_DEBUG "%s: uba_suspend_agetty_process stopped\n", UBA_MODULE_NAME);
        return 0;
}

#define UBA_VENDOR_ID  0x0781
#define UBA_PRODUCT_ID 0x5591
#define UBA_SERIAL "4C530001301105102492"

static int uba_probe(struct usb_interface *interface, const struct usb_device_id *id)
{
        struct usb_device *dev;

        dev = interface_to_usbdev(interface);
        printk(KERN_NOTICE "%s: USB flash drive plugged\n", UBA_MODULE_NAME);
        printk(KERN_NOTICE "%s: VID: 0x%04X, PID: 0x%04X, Serial: %s\n", UBA_MODULE_NAME,
            id->idVendor, id->idProduct, dev->serial);

        if (!program_has_terminated) {
                if (strcmp(dev->serial, UBA_SERIAL)) {
                        printk(KERN_NOTICE "%s: The serial of the USB flash drive doesn't
                            match\n", UBA_MODULE_NAME);
                        return -1;
                }

                uba_stop_thread(&suspend_agetty_process);
                uba_set_task_state("agetty", TASK_INTERRUPTIBLE);

                printk(KERN_NOTICE "%s: USB boot authentication completed successfully\n",
                    UBA_MODULE_NAME);
                program_has_terminated = true;
        }

        return 0;
}

static void uba_disconnect(struct usb_interface *interface)
{
        printk(KERN_DEBUG "%s: USB flash drive unplugged\n", UBA_MODULE_NAME);
}

static struct usb_device_id uba_table[] = {
        { USB_DEVICE(UBA_VENDOR_ID, UBA_PRODUCT_ID) },
        { },
};

MODULE_DEVICE_TABLE(usb, uba_table);

static struct usb_driver uba_driver = {
        .name = UBA_MODULE_NAME,
        .probe = uba_probe,
        .disconnect = uba_disconnect,
        .id_table = uba_table,
};

static int __init usb_boot_authentication_init(void)
{
        int rc;

        program_has_terminated = false;
```

```
113
114         if ((rc = usb_register(&uba_driver)) < 0) {
115                 printk(KERN_ERR "%s: usb_register failed with code %d\n", UBA_MODULE_NAME, rc);
116                 return rc;
117         }
118
119         if (IS_ERR(suspend_agetty_process = kthread_run(uba_suspend_agetty_process, NULL,
                "uba_suspend_agetty_process"))) {
120                 rc = (int) PTR_ERR(suspend_agetty_process);
121                 printk(KERN_ERR "%s: kthread_run failed with code %d\n", UBA_MODULE_NAME, rc);
122                 return rc;
123         }
124
125         printk(KERN_DEBUG "%s: module loaded\n", UBA_MODULE_NAME);
126         return 0;
127 }
128
129 static void __exit usb_boot_authentication_exit(void)
130 {
131         if (suspend_agetty_process) {
132                 uba_stop_thread(&suspend_agetty_process);
133         }
134
135         usb_deregister(&uba_driver);
136
137         printk(KERN_DEBUG "%s: module unloaded\n", UBA_MODULE_NAME);
138 }
139
140 module_init(usb_boot_authentication_init)
141 module_exit(usb_boot_authentication_exit)
```

Приложение B. Makefile

Листинг 3.4: `Makefile`

```
SHELL = /bin/bash

UBA_MODULE_NAME = usb_boot_authentication

obj-m += $(UBA_MODULE_NAME).o
UBA_LKM = $(UBA_MODULE_NAME).ko

KERNEL_DIR = /lib/modules/$(shell uname -r)
KERNEL_BUILD_DIR = $(KERNEL_DIR)/build
KERNEL_DRIVERS_UBA_DIR = $(KERNEL_DIR)/kernel/drivers/$(UBA_MODULE_NAME)

GRUB_CONFIG = /etc/default/grub
GRUB_CONFIG_BACKUP = $(GRUB_CONFIG).backup

RSYSLOG_CONFIG = /etc/rsyslog.d/10-usb-boot-authentication.conf


.PHONY: all default install modules modules_install help clean \
        enable_at_boot \
        disable_at_boot \
        boot_in_console_mode \
        boot_in_gui_mode \
        enable_printing_kernel_journal_on_tty \
        disable_printing_kernel_journal_on_tty


all default: modules
install: modules_install


modules modules_install help clean:
        make -C $(KERNEL_BUILD_DIR) M=$(shell pwd) $@


enable_at_boot: modules $(KERNEL_DRIVERS_UBA_DIR)
ifneq ($(shell grep $(UBA_MODULE_NAME) /etc/modules),)
        $(error module already loaded)
endif

        cp $(UBA_LKM) $(KERNEL_DRIVERS_UBA_DIR)
        echo $(UBA_MODULE_NAME) >> /etc/modules
        depmod


disable_at_boot:
ifeq ($(shell grep $(UBA_MODULE_NAME) /etc/modules),)
        $(error module already unloaded)
endif

ifneq ($(shell lsmod | grep $(UBA_MODULE_NAME)),)
        rmmod $(UBA_LKM)
endif

        rm -f $(KERNEL_DRIVERS_UBA_DIR)/$(UBA_LKM)
        rmdir $(KERNEL_DRIVERS_UBA_DIR)
```

```
56          sed --in-place '/$(UBA_MODULE_NAME)/d' /etc/modules
57          depmod
58
59
60  boot_in_console_mode: $(GRUB_CONFIG_BACKUP)
61          sed --in-place 's/^GRUB_CMDLINE_LINUX_DEFAULT=\(.*\)$$/#GRUB_CMDLINE_LINUX_DEFAULT=\1/'
                $(GRUB_CONFIG)
62          sed --in-place 's/^GRUB_CMDLINE_LINUX=.*$$/GRUB_CMDLINE_LINUX=text/' $(GRUB_CONFIG)
63          sed --in-place 's/^#GRUB_TERMINAL=.*$$/GRUB_TERMINAL=console/' $(GRUB_CONFIG)
64          update-grub
65          systemctl set-default multi-user.target
66
67
68  boot_in_graphical_mode:
69          mv $(GRUB_CONFIG_BACKUP) $(GRUB_CONFIG)
70          update-grub
71          systemctl set-default graphical.target
72
73
74  enable_printing_kernel_journal_on_tty:
75          echo 'kern.notice /dev/tty1' > $(RSYSLOG_CONFIG)
76
77
78  disable_printing_kernel_journal_on_tty:
79          rm -f $(RSYSLOG_CONFIG)
80
81
82  $(KERNEL_DRIVERS_UBA_DIR):
83          mkdir -p $@
84
85
86  $(GRUB_CONFIG_BACKUP):
87          cp --no-clobber $(GRUB_CONFIG) $@
```